.NET Full Stack Development Program

LINQ & Thread

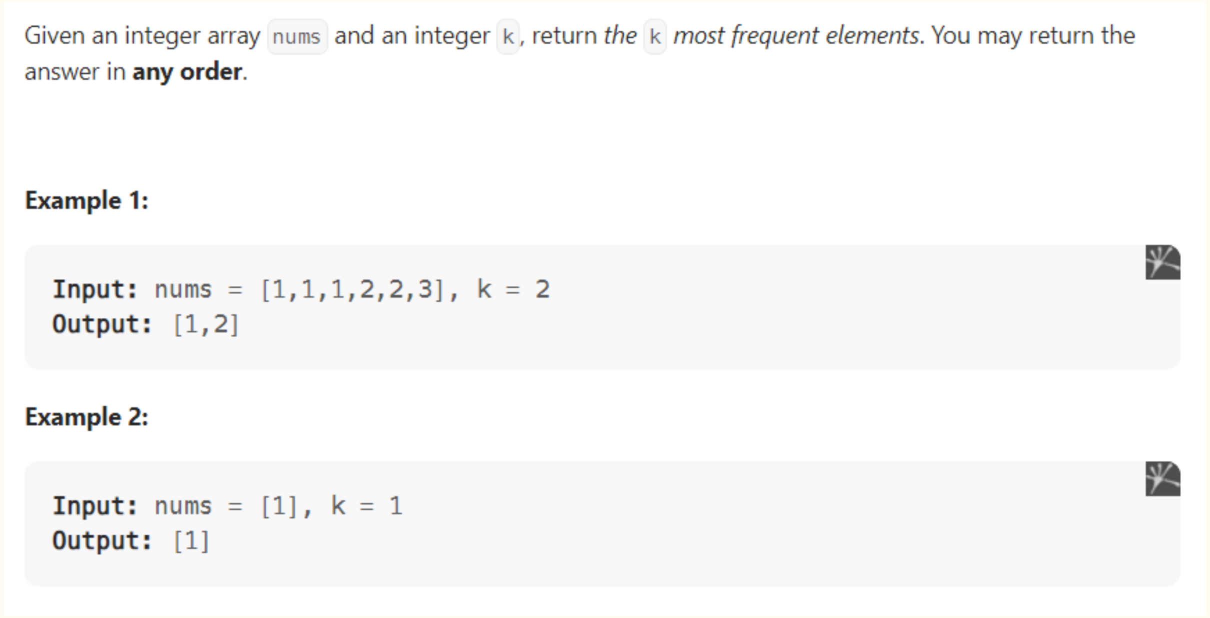
Outline

- LINQ
  - What is LINQ?
  - LINQ to Objects
  - Standard Query Operators
  - LINQ to SQL

- Thread
  - Thread & Process
  - Thread Class
  - Thread Life Cycle
  - Thread Problem / Thread Safety
  - Thread Synchronization
  - Dead Lock

Question

Top K Frequent Elements



Given an integer array `nums` and an integer `k`, return *the* `k` *most frequent elements*. You may return the answer in **any order**.

Example 1:

```
Input: nums = [1,1,1,2,2,3], k = 2
Output: [1,2]
```

Example 2:

```
Input: nums = [1], k = 1
Output: [1]
```

LINQ

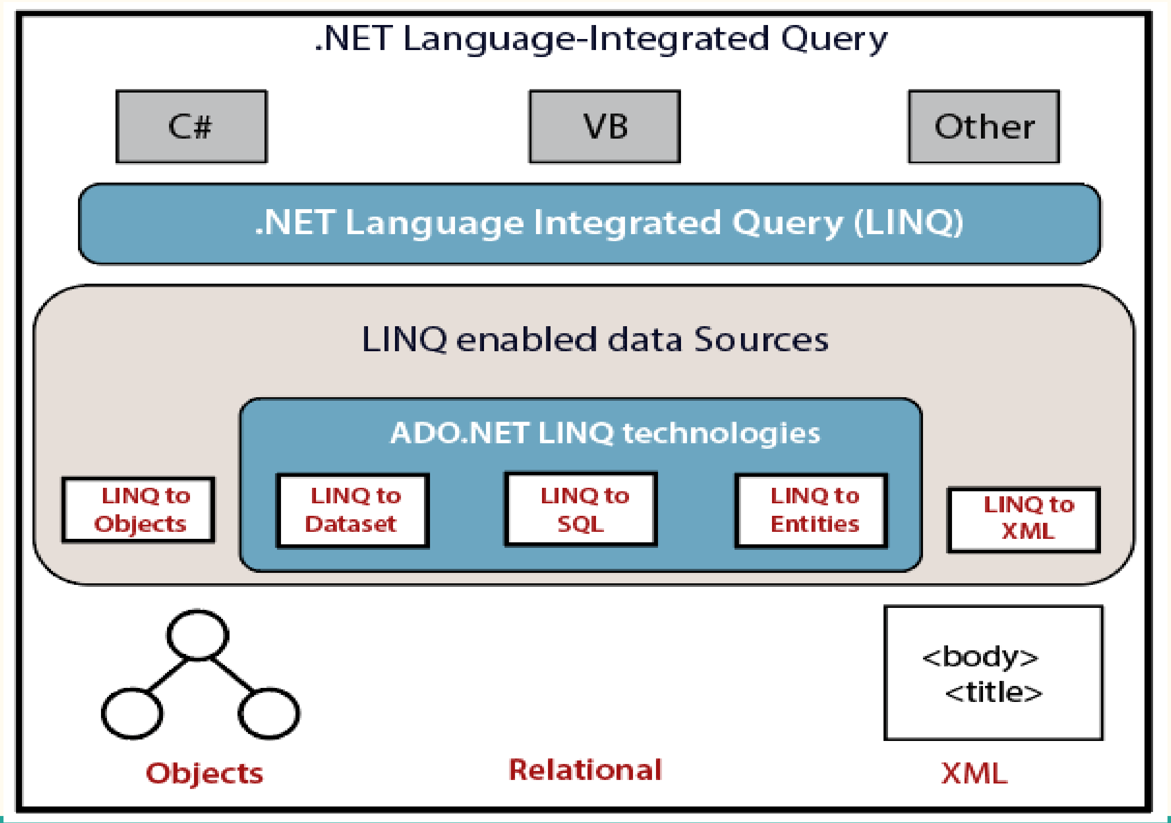
What is LINQ?

- LINQ(Language Integrated Query) provides us with common query syntax which allows us to query the data from various data sources.

- It was introduced by Microsoft with .NET Framework 3.5 and C# 3.0 and is available in **System.Linq** namespace.

- It supports a consistent query experience(LINQ Provider)
    - LINQ to objects
    - LINQ to SQL
    - LINQ to XML
    - …

How does LINQ work?


.NET Language-Integrated Query
C#
VB
Other
.NET Language Integrated Query (LINQ)
LINQ enabled data Sources
ADO.NET LINQ technologies
LINQ to Objects
LINQ to Dataset
LINQ to SQL
LINQ to Entities
LINQ to XML
<body>
<title>
Objects
Relational
XML

LINQ Providers

- ❑ **LINQ to objects** (No provider needed): allows us to query in-memory objects from an array, collection and generics types.
- **LINQ to XML**(XLINQ): works with XML documents.
- **LINQ to SQL**(DLINQ): works with the SQL Server database.
- **LINQ to Datasets**: provides us the flexibility to query data cached in a Dataset.
- **LINQ to Entities**: is used to query any database(including SQL Server, Oracle, MySQL etc.)

LINQ to Objects

The term "LINQ to Objects" refers to the use of LINQ queries with any IEnumerable or IEnumerable<T> collection directly, without the use of an intermediate LINQ provider or API such as LINQ to SQL or LINQ to XML. You can use LINQ to query any enumerable collections such as List<T>, Array, or Dictionary<TKey,TValue>. The collection may be user-defined.

Advantages:

- They are more concise and readable, especially when filtering multiple conditions.
- They provide powerful filtering, ordering, and grouping capabilities with a minimum of application code.
- They can be ported to other data sources with little or no modification.

IEnumerable

- IEnumerable is **an interface that defines one method, GetEnumerator() which returns an IEnumerator type**.

- The IEnumerable interface is a type of iteration design pattern. It means we can **iterate on** the collection of the type IEnumerable.

- Basically, you can write LINQ queries for any type that supports IEnumerable or the generic IEnumerable<T> interface.

Different ways to write a LINQ

- Query Syntax
- Method Syntax
- Mixed Syntax (Query + Method)

```
form data in datasource
where condition
select data;
```

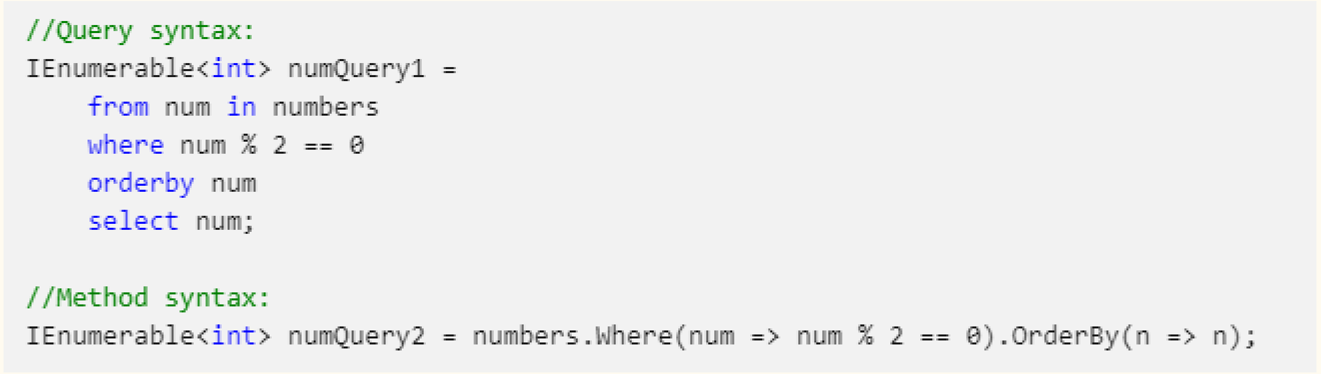
```
DataSource.ConditionMethod().OtherMethod();
```

```
(form data in datasource
where condition
select data).Method();
```

Query Syntax vs. Method Syntax

Most queries in the introductory Language Integrated Query (LINQ) documentation are written by using the LINQ declarative query syntax. However, the query syntax must be translated into method calls for the .NET common language runtime (CLR) when the code is compiled. These method calls invoke the standard query operators, which have names such as **Where, Select, GroupBy, Join, Max, and Average**. You can call them directly by using **method syntax** instead of query syntax.

There will be some queries that must be expressed as method calls. you must use a method call to express a query that retrieves the number of elements that match a specified condition. You also must use a method call for a query that retrieves the element that has the maximum value in a source sequence.

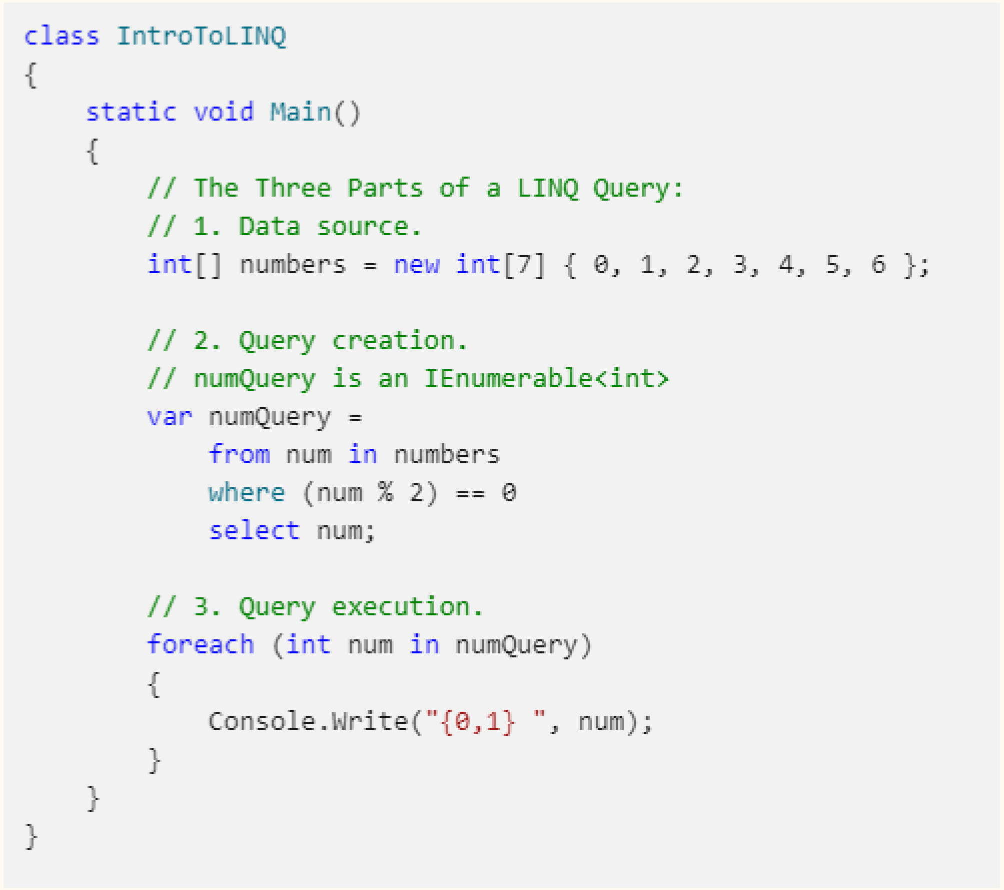


```
//Query syntax:
IEnumerable<int> numQuery1 =
    from num in numbers
    where num % 2 == 0
    orderby num
    select num;

//Method syntax:
IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 == 0).OrderBy(n => n);
```

LINQ Queries

- All LINQ query operations consist of three distinct actions:

    1. Obtain the data source
    2. Create the query
    3. Execute the query

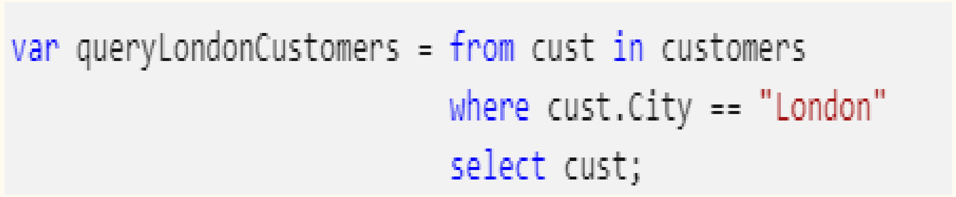


```csharp
class IntroToLINQ
{
    static void Main()
    {
        // The Three Parts of a LINQ Query:
        // 1. Data source.
        int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

        // 2. Query creation.
        // numQuery is an IEnumerable<int>
        var numQuery =
            from num in numbers
            where (num % 2) == 0
            select num;

        // 3. Query execution.
        foreach (int num in numQuery)
        {
            Console.Write("{0,1} ", num);
        }
    }
}
```

Create the Query

The query expression contains **three clauses**: from, where and select

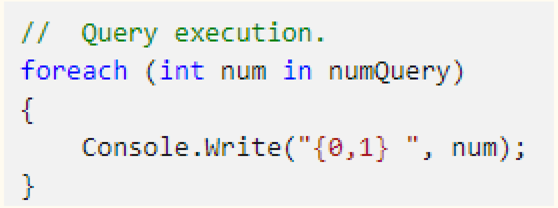
The **from** clause specifies the data source, the **where** clause applies the filter, and the **select** clause specifies the type of the returned elements.



```
var queryLondonCustomers = from cust in customers
                           where cust.City == "London"
                           select cust;
```

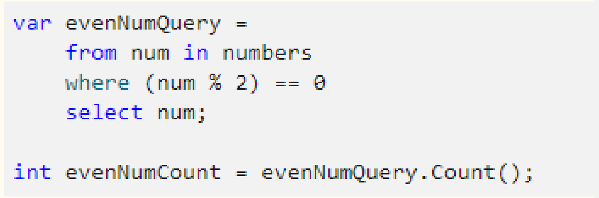
Execute the Query

The actual execution of the query is **deferred** until you iterate over the query variable in a foreach statement.

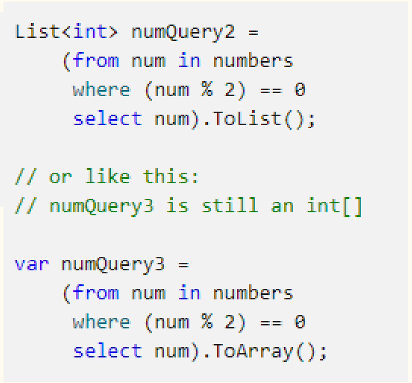


```
//  Query execution.
foreach (int num in numQuery)
{
    Console.Write("{0,1} ", num);
}
```

Manners Of Execution



```
var evenNumQuery =
    from num in numbers
    where (num % 2) == 0
    select num;

int evenNumCount = evenNumQuery.Count();
```



```
List<int> numQuery2 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToList();

// or like this:
// numQuery3 is still an int[]

var numQuery3 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToArray();
```

LINQ operators are divided into 2 categories:

- Deferred Execution

  Deferred execution means that the operation is not performed at the point in the code where the query is declared. The operation is performed only when the query variable is enumerated, for example by using a foreach statement.

  Almost all the standard query operators whose return type is IEnumerable<T>

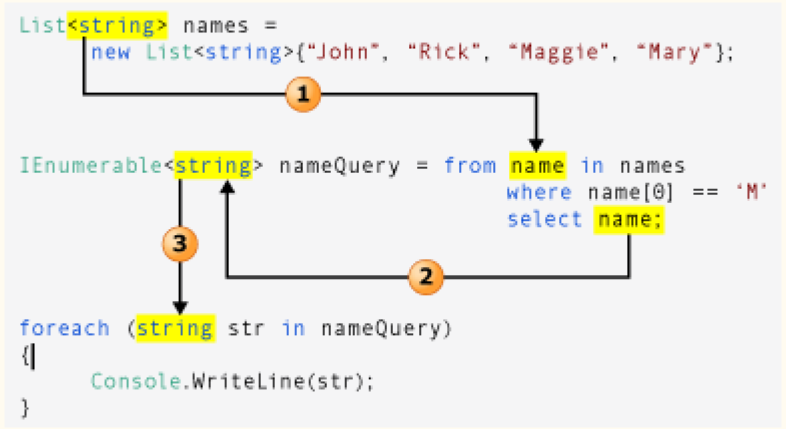
  Ex: Select(), SelectMany(), Where(), etc.

- Immediate Execution

  Immediate execution means that the data source is read and the operation is performed at the point in the code where the query is declared.

  Ex: ToArray(), ToList(), Aggregate Methods: etc.

Type Relationships in LINQ



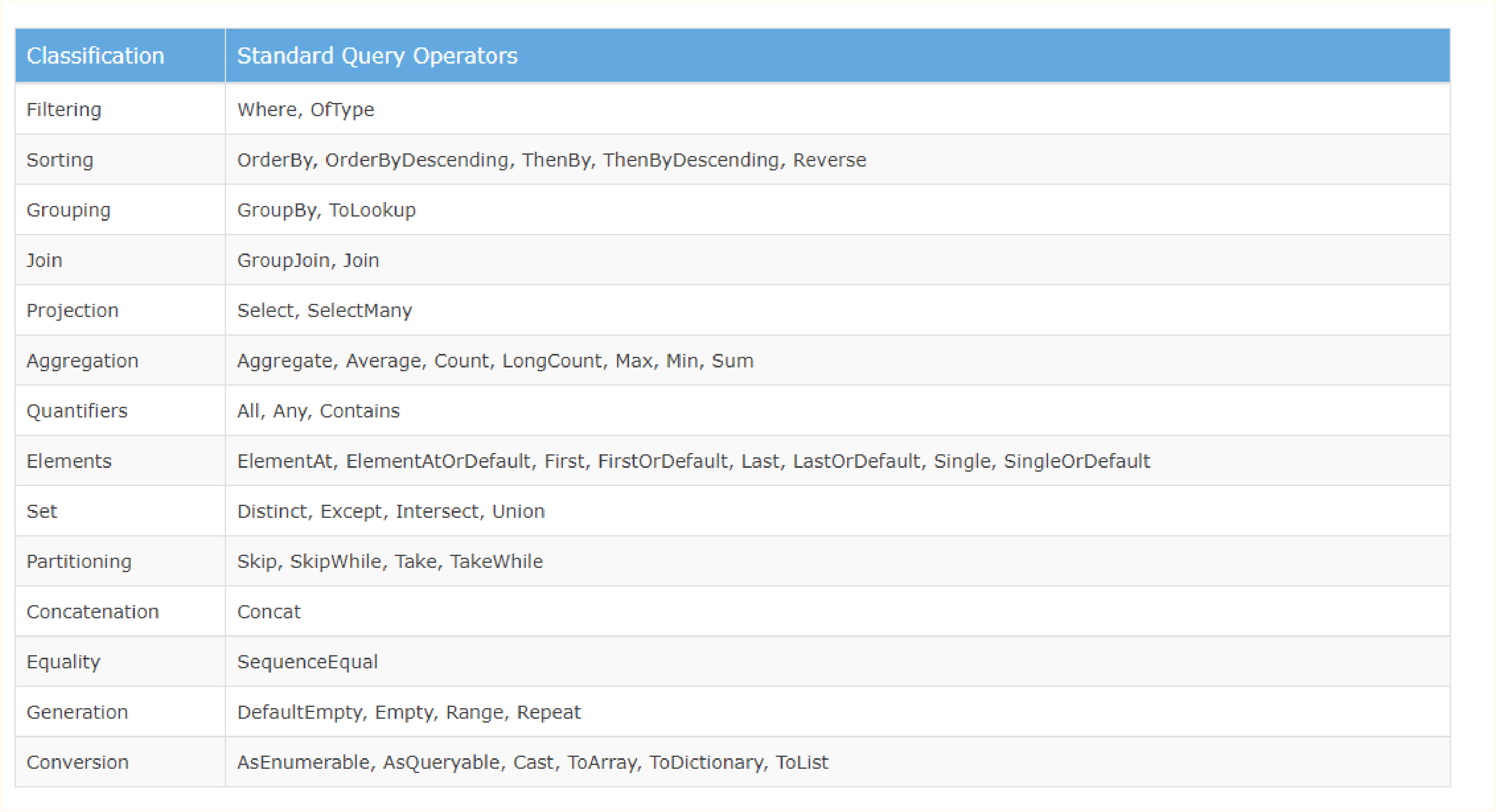
```
List<string> names =
      new List<string>{"John", "Rick", "Maggie", "Mary"};

IEnumerable<string> nameQuery = from name in names
                                where name[0] == 'M'
                                select name;

foreach (string str in nameQuery)
{
      Console.WriteLine(str);
}
```

Standard LINQ Operators

Standard Query Operators

- The ***standard query operators*** are the **methods** that form the LINQ pattern. Most of these methods operate on sequences, where a sequence is an object whose type implements the IEnumerable<T> interface. The standard query operators provide query capabilities including **filtering, projection, aggregation, sorting and more.**

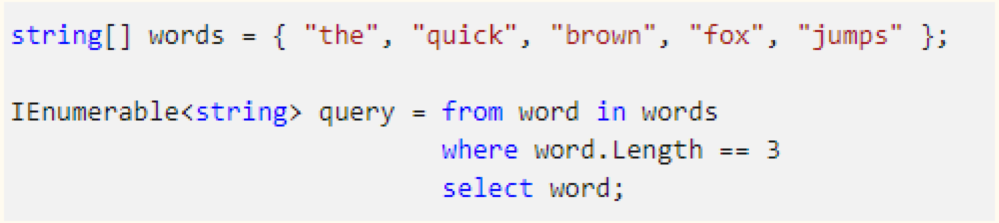
- Query Expression Syntax for Standard Query Operators:
  - Cast, GroupBy, GroupJoin, Join, OrderBy, OrderByDescending, Select, SelectMany, ThenBy, ThenByDescending, Where, and more



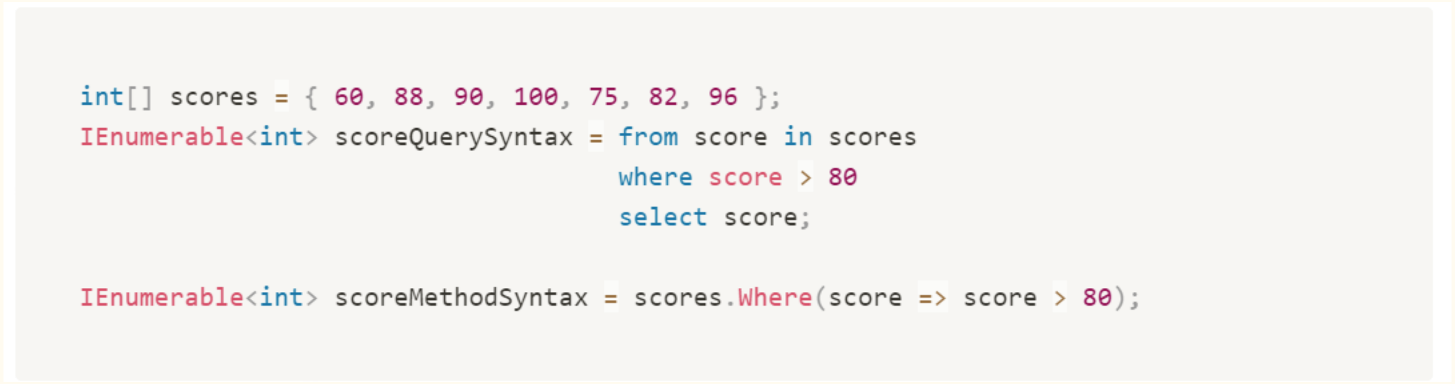
Classification	Standard Query Operators
Filtering	Where, OfType
Sorting	OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse
Grouping	GroupBy, ToLookup
Join	GroupJoin, Join
Projection	Select, SelectMany
Aggregation	Aggregate, Average, Count, LongCount, Max, Min, Sum
Quantifiers	All, Any, Contains
Elements	ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault
Set	Distinct, Except, Intersect, Union
Partitioning	Skip, SkipWhile, Take, TakeWhile
Concatenation	Concat
Equality	SequenceEqual
Generation	DefaultEmpty, Empty, Range, Repeat
Conversion	AsEnumerable, AsQueryable, Cast, ToArray, ToDictionary, ToList

Filtering Data

Filtering refers to the operation of restricting the result set to contain only those elements that satisfy a specified condition



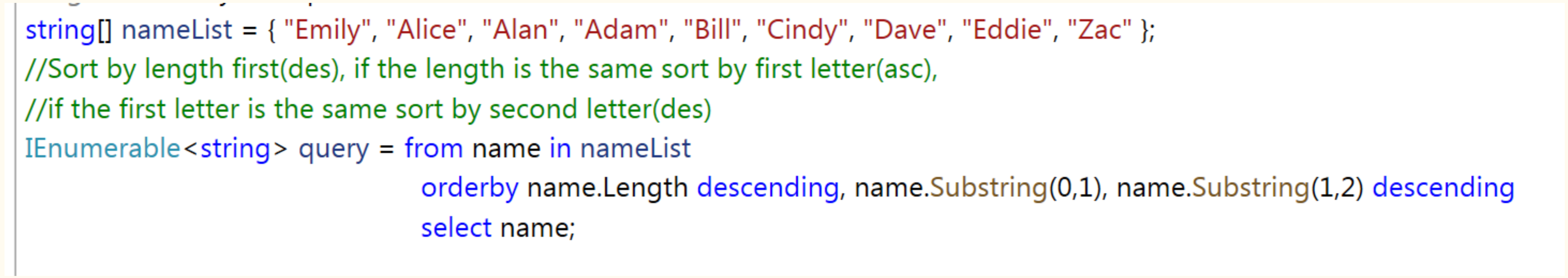
```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                            where word.Length == 3
                            select word;
```



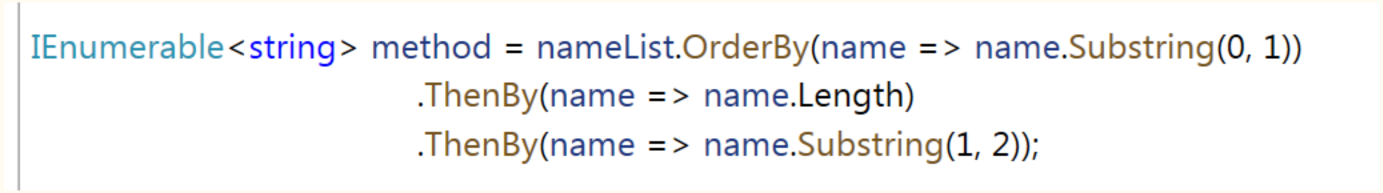
```
int[] scores = { 60, 88, 90, 100, 75, 82, 96 };
IEnumerable<int> scoreQuerySyntax = from score in scores
                                    where score > 80
                                    select score;

IEnumerable<int> scoreMethodSyntax = scores.Where(score => score > 80);
```

Sorting Data

Sorting operations order the elements of a sequence based on one or more attributes:
OrderBy() | OrderByDescending() | ThenBy() | ThenByDescending()



```
string[] nameList = { "Emily", "Alice", "Alan", "Adam", "Bill", "Cindy", "Dave", "Eddie", "Zac" };
//Sort by length first(des), if the length is the same sort by first letter(asc),
//if the first letter is the same sort by second letter(des)
IEnumerable<string> query = from name in nameList
                            orderby name.Length descending, name.Substring(0,1), name.Substring(1,2) descending
                            select name;
```



```
IEnumerable<string> method = nameList.OrderBy(name => name.Substring(0, 1))
                             .ThenBy(name => name.Length)
                             .ThenBy(name => name.Substring(1, 2));
```

Set Operators

- Distinct

```
IEnumerable<string> query = from planet in planets.Distinct()
                            select planet;
```

Removes duplicate values from a collection.

- Except

```
IEnumerable<string> query = from planet in planets1.Except(planets2)
                            select planet;
```

Returns the set difference, which means the elements of one collection that do not appear in a second collection.

- Intersect

```
IEnumerable<string> query = from planet in planets1.Intersect(planets2)
                            select planet;
```

Returns the set intersection, which means elements that appear in each of two collections

- Union

```
IEnumerable<string> query = from planet in planets1.Union(planets2)
                            select planet;
```

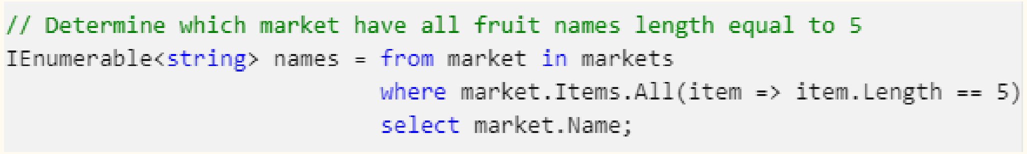
Returns the set union, which means unique elements that appear in either of two collections

Quantifier Operation

Quantifier operations return a Boolean value that indicates whether some or all of the elements in a sequence satisfy a condition

- All

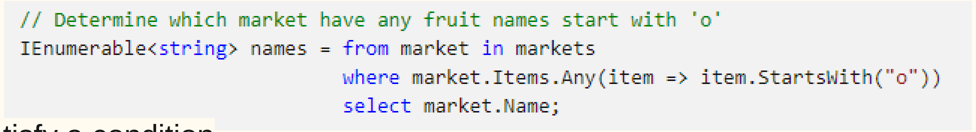
  Determines whether all the elements in a sequence satisfy a condition.



```
// Determine which market have all fruit names length equal to 5
IEnumerable<string> names = from market in markets
                            where market.Items.All(item => item.Length == 5)
                            select market.Name;
```

- Any

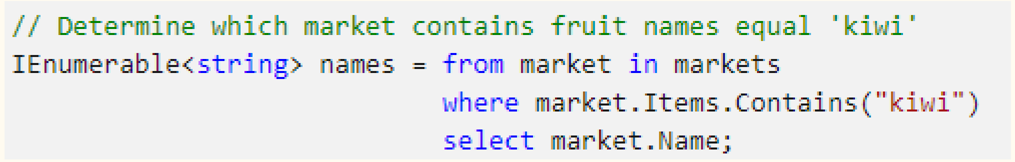
  Determines whether any elements in a sequence satisfy a condition



```
// Determine which market have any fruit names start with 'o'
IEnumerable<string> names = from market in markets
                            where market.Items.Any(item => item.StartsWith("o"))
                            select market.Name;
```

- Contains

  Determines whether a sequence contains a specified element



```
// Determine which market contains fruit names equal 'kiwi'
IEnumerable<string> names = from market in markets
                            where market.Items.Contains("kiwi")
                            select market.Name;
```

Projection Operation

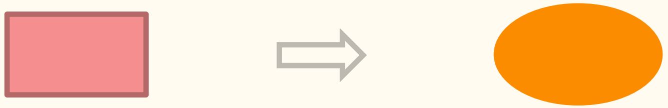
Projection refers to the operation of transforming an object into a new form that often consists only of those properties that will be subsequently used.

- Select

  Projects values that are based on a transform function

- SelectMany

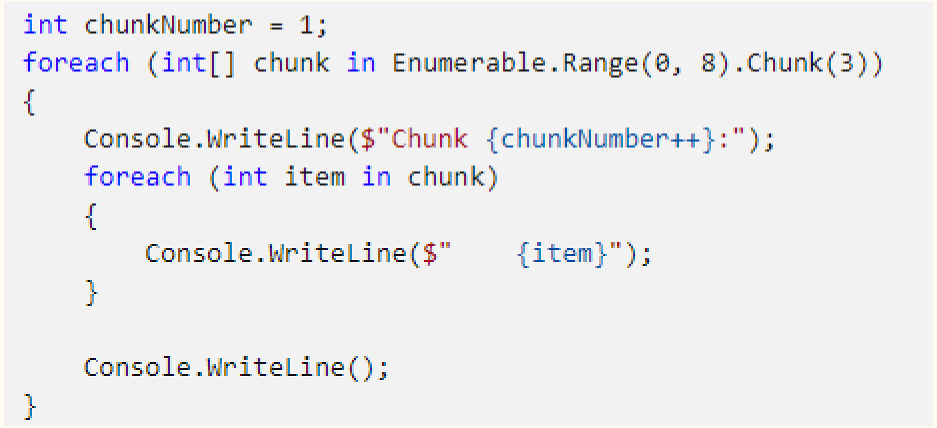
  Projects sequences of values that are based on a transform function and then flattens them into one sequence



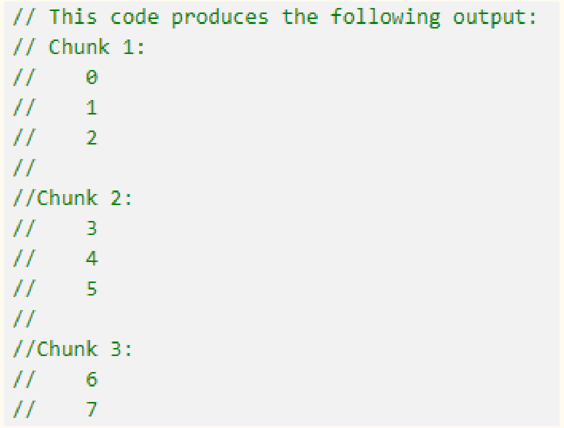
Partitioning Data

Partitioning in LINQ refers to the operation of dividing an input sequence into two sections, without rearranging the elements, and then returning one of the sections

- Skip: Skips elements up to a specified position in a sequence
- SkipWhile: Skips elements based on a predicate function until an element does not satisfy the condition
- Take: Takes elements up to a specified position in a sequence
- TakeWhile: Takes elements based on a predicate function until an element does not satisfy the condition
- Chunk: Splits the elements of a sequence into chunks of a specified maximum size



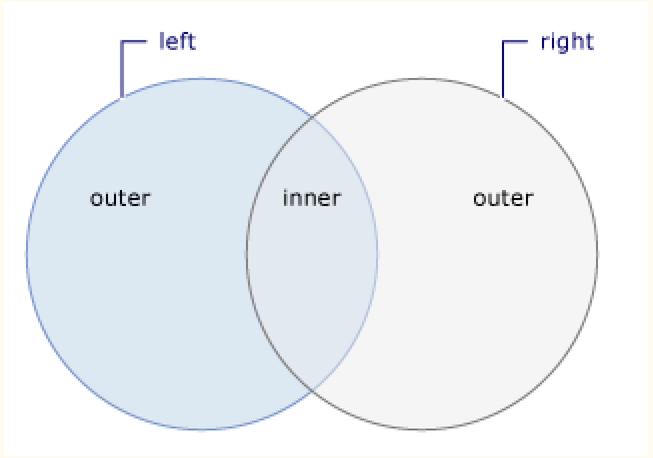
```
int chunkNumber = 1;
foreach (int[] chunk in Enumerable.Range(0, 8).Chunk(3))
{
    Console.WriteLine($"Chunk {chunkNumber++}:");
    foreach (int item in chunk)
    {
        Console.WriteLine($"    {item}");
    }

    Console.WriteLine();
}
```



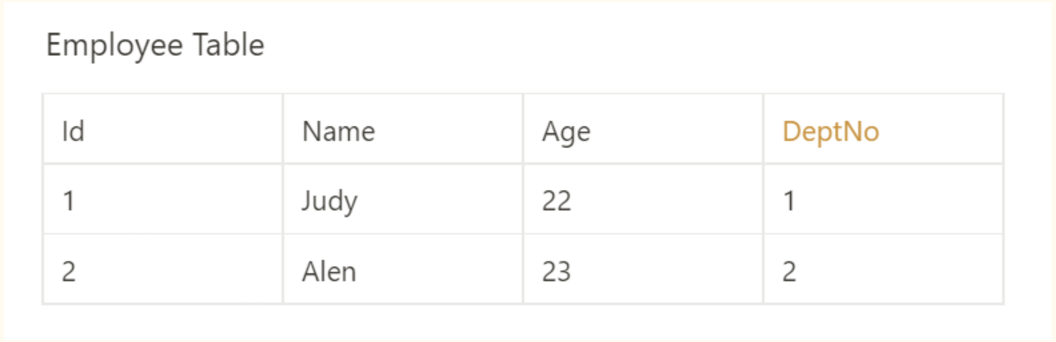
```
// This code produces the following output:
// Chunk 1:
//    0
//    1
//    2
//
//Chunk 2:
//    3
//    4
//    5
//
//Chunk 3:
//    6
//    7
```

Join Operation

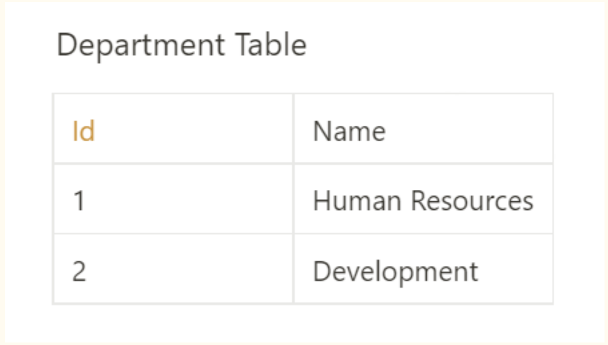
A ***join*** of two data sources is the association of objects in one data source with objects that share a common attribute in another data source.



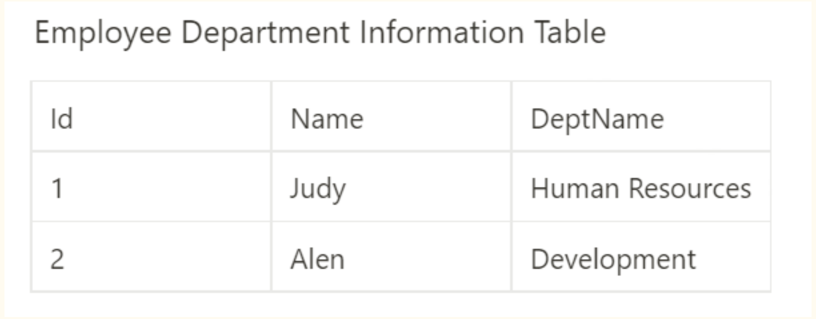
Join Operator


Employee Table

Id	Name	Age	DeptNo
1	Judy	22	1
2	Alen	23	2

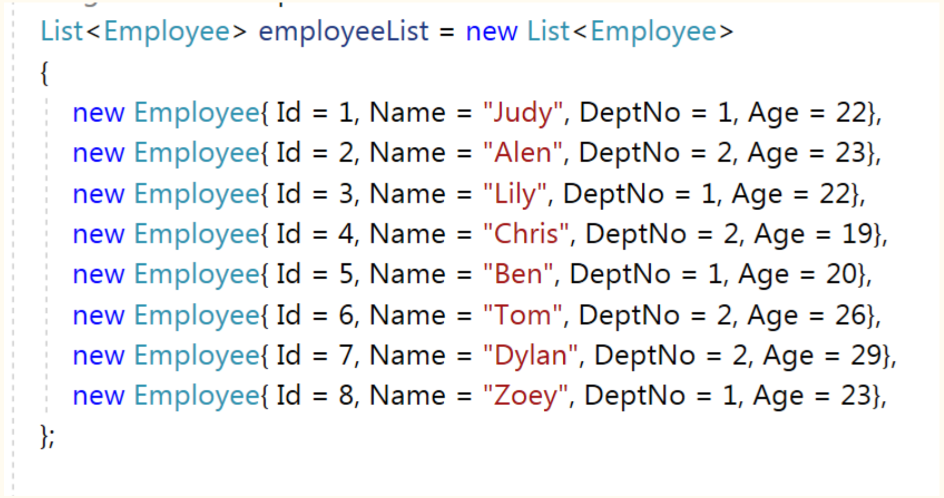

Department Table

Id	Name
1	Human Resources
2	Development

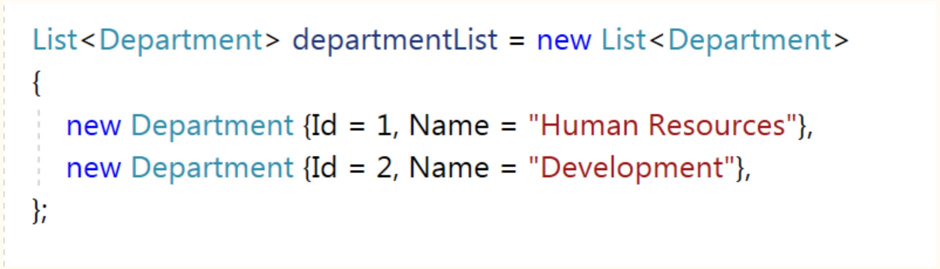

Employee Department Information Table

Id	Name	DeptName
1	Judy	Human Resources
2	Alen	Development

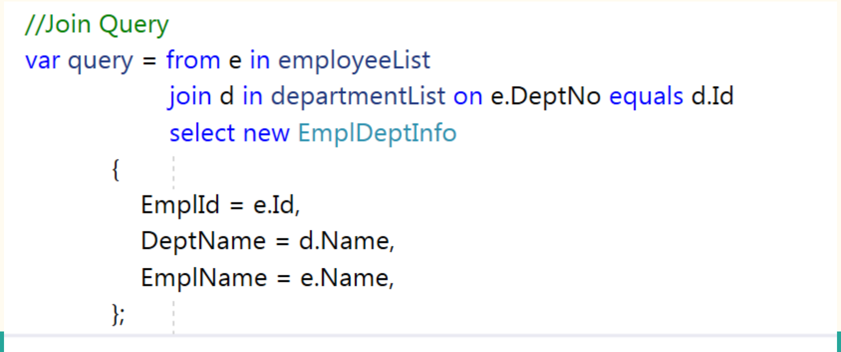
Join Operation



```
List<Employee> employeeList = new List<Employee>
{
    new Employee{ Id = 1, Name = "Judy", DeptNo = 1, Age = 22},
    new Employee{ Id = 2, Name = "Alen", DeptNo = 2, Age = 23},
    new Employee{ Id = 3, Name = "Lily", DeptNo = 1, Age = 22},
    new Employee{ Id = 4, Name = "Chris", DeptNo = 2, Age = 19},
    new Employee{ Id = 5, Name = "Ben", DeptNo = 1, Age = 20},
    new Employee{ Id = 6, Name = "Tom", DeptNo = 2, Age = 26},
    new Employee{ Id = 7, Name = "Dylan", DeptNo = 2, Age = 29},
    new Employee{ Id = 8, Name = "Zoey", DeptNo = 1, Age = 23},
};
```

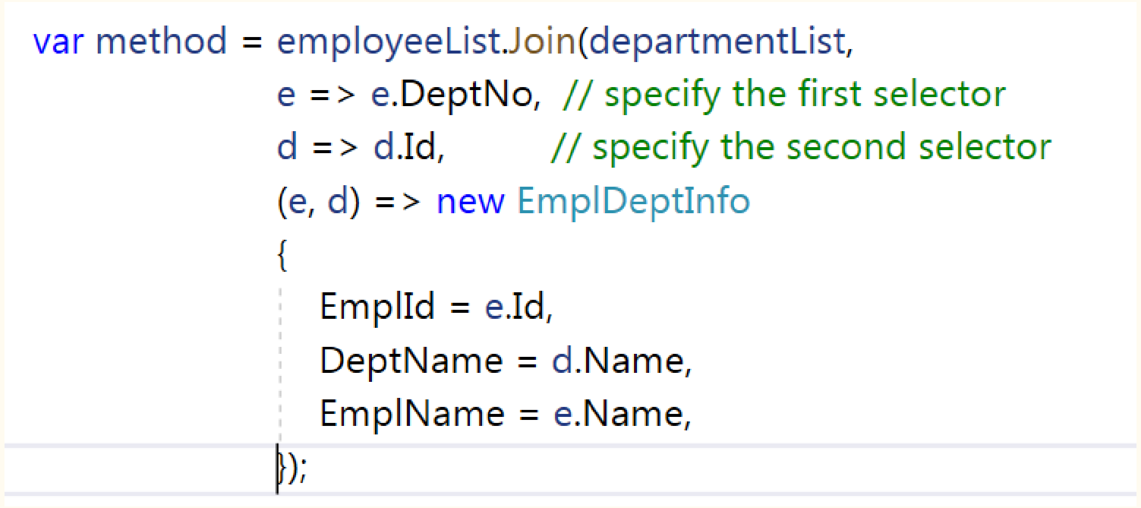


```
List<Department> departmentList = new List<Department>
{
    new Department {Id = 1, Name = "Human Resources"},
    new Department {Id = 2, Name = "Development"},
};
```



```
//Join Query
var query = from e in employeeList
            join d in departmentList on e.DeptNo equals d.Id
            select new EmplDeptInfo
        {
            EmplId = e.Id,
            DeptName = d.Name,
            EmplName = e.Name,
        };
```

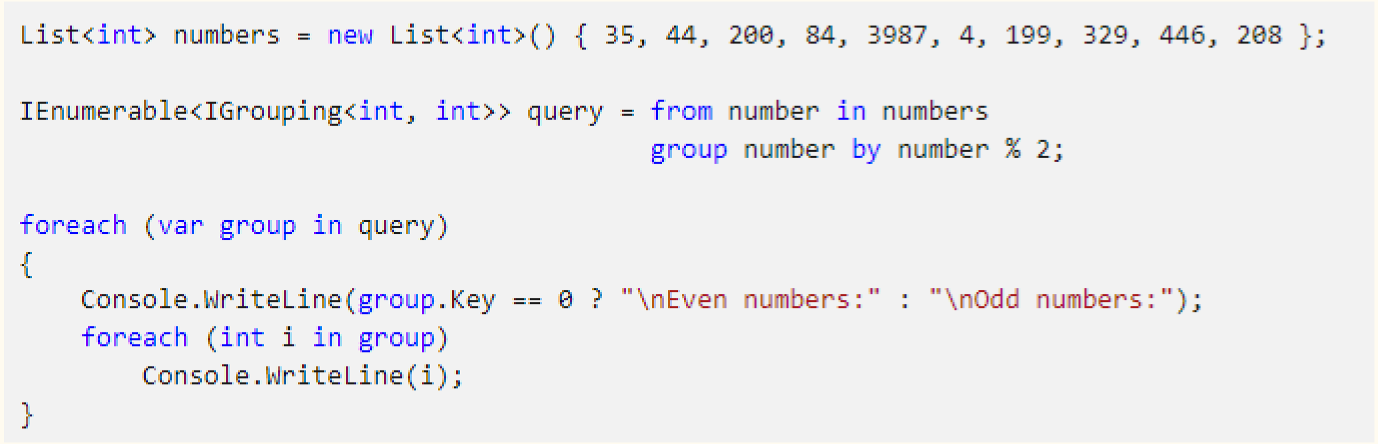
Join Operation



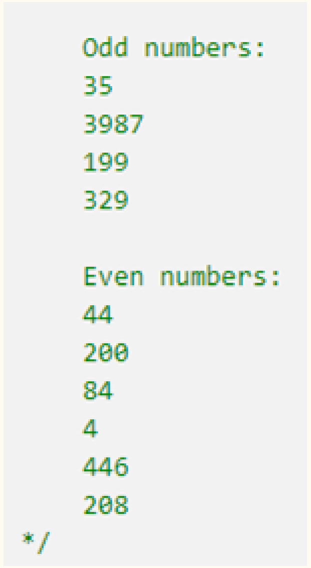
```
var method = employeeList.Join(departmentList,
                        e => e.DeptNo,  // specify the first selector
                        d => d.Id,        // specify the second selector
                        (e, d) => new EmplDeptInfo
                        {
                            EmplId = e.Id,
                            DeptName = d.Name,
                            EmplName = e.Name,
                        });
```

Grouping Data

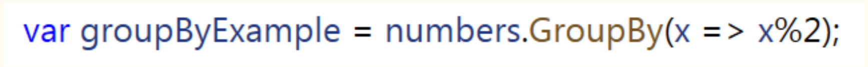
- GroupBy()/group by: takes a flat sequence of elements and then organizes the elements into groups, based on a given key, it will return an **IEnumerable<IGrouping<TKey, TSource>>** where **TKey** is nothing but the **Key** value on which the grouping has been formed and **TSource** is the collection of elements that matches the grouping key value.



```
List<int> numbers = new List<int>() { 35, 44, 200, 84, 3987, 4, 199, 329, 446, 208 };

IEnumerable<IGrouping<int, int>> query = from number in numbers
                                         group number by number % 2;

foreach (var group in query)
{
    Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd numbers:");
    foreach (int i in group)
        Console.WriteLine(i);
}
```



```
    Odd numbers:
    35
    3987
    199
    329

    Even numbers:
    44
    200
    84
    4
    446
    208
*/
```



```
var groupByExample = numbers.GroupBy(x => x%2);
```

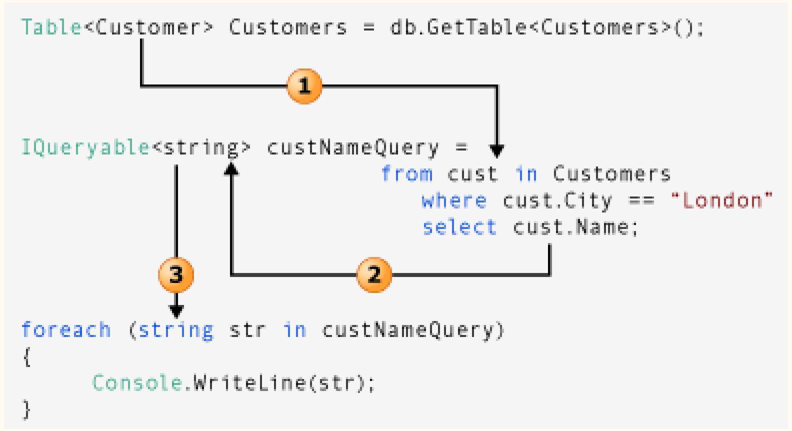
Aggregate Methods

- **Sum():** This method is used to calculate the total(sum) value of the collection.
- **Max():** This method is used to find the largest value in the collection.
- **Min():** This method is used to find the smallest value in the collection.
- **Average():** This method is used to calculate the average value(*double*) of the numeric type of the collection.
- **Count():** This method is used to count the number of elements present in the collection.

```
int[] arr = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int sum = arr.Sum();
int max = arr.Max();
int min = arr.Min();
double avg = arr.Average();
int count = arr.Count();
```

LINQ to SQL

LINQ to SQL



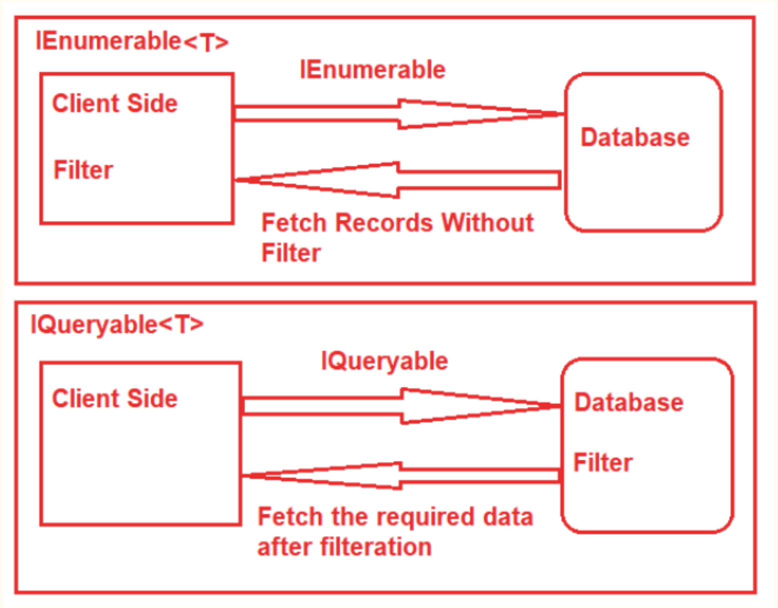
```
Table<Customer> Customers = db.GetTable<Customers>();

IQueryable<string> custNameQuery =
                    from cust in Customers
                        where cust.City == "London"
                        select cust.Name;

foreach (string str in custNameQuery)
{
       Console.WriteLine(str);
}
```

IQueryable

- The IQueryable interface inherits the IEnumerable interface so that if it represents a query, the results of that query can be enumerated.

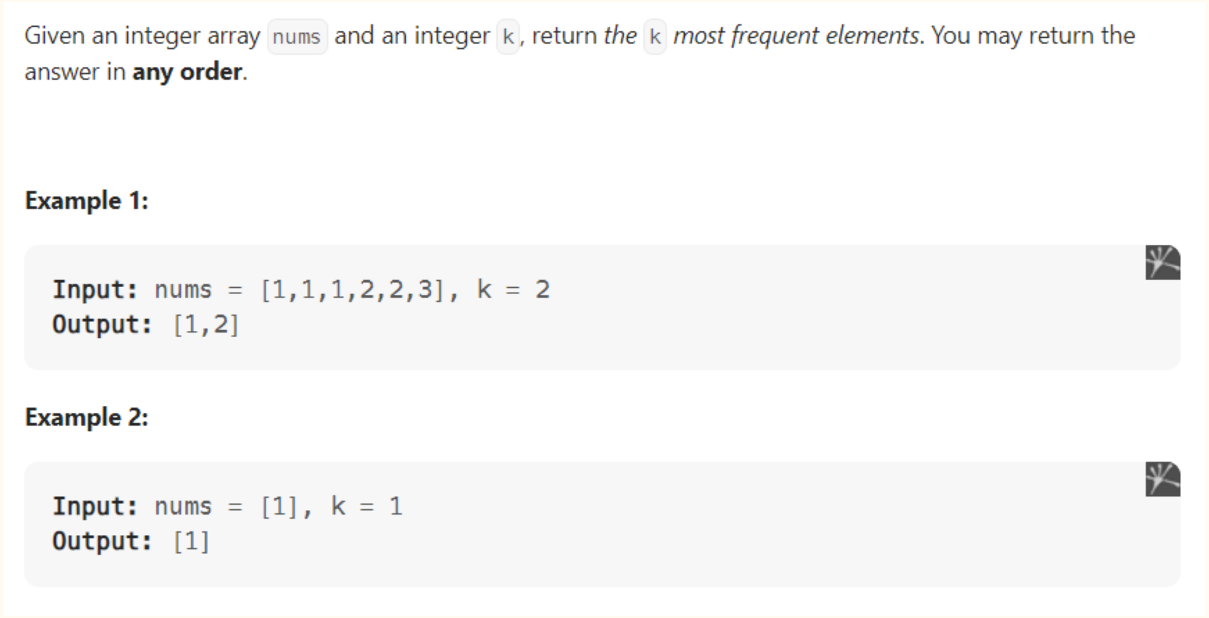
- We can use AsQueryable() and AsEnumerable() to convert them.

IEnumerable vs IQueryable



Question

Top K Frequent Elements



Given an integer array `nums` and an integer `k`, return *the* `k` *most frequent elements*. You may return the answer in **any order**.

Example 1:

```
Input: nums = [1,1,1,2,2,3], k = 2
Output: [1,2]
```

Example 2:

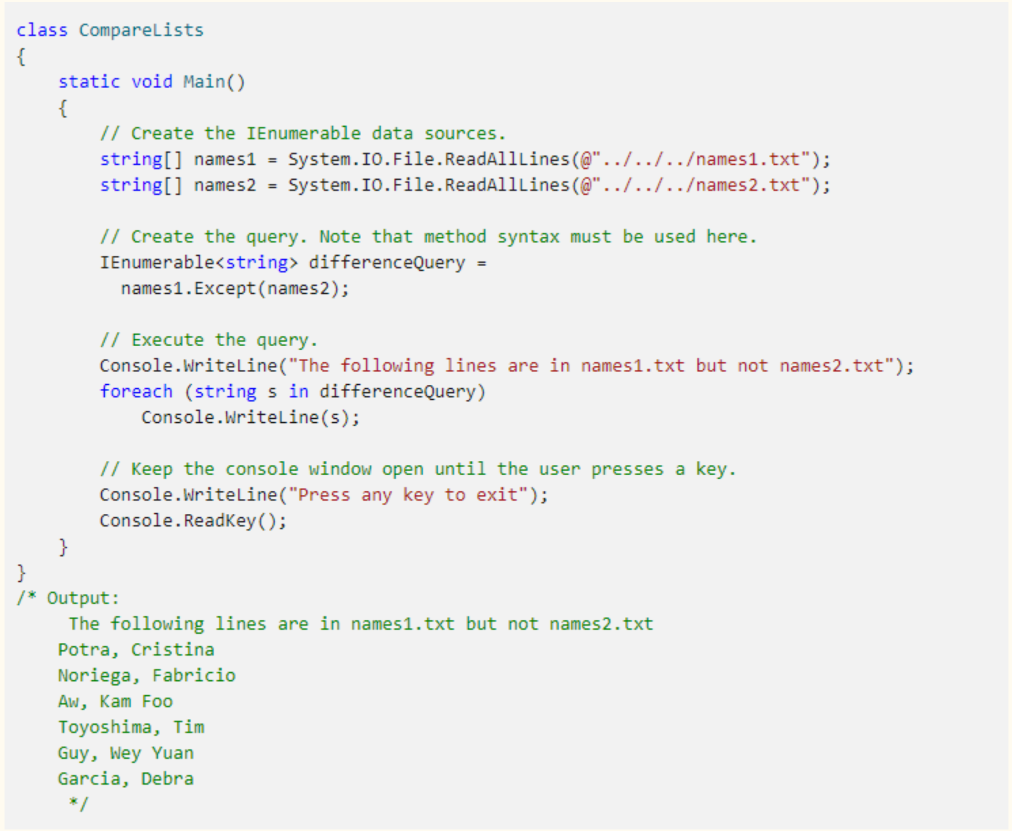
```
Input: nums = [1], k = 1
Output: [1]
```

Casting

```
class Plant
{
    public string Name { get; set; }
}

class CarnivorousPlant : Plant
{
    public string TrapType { get; set; }
}
```

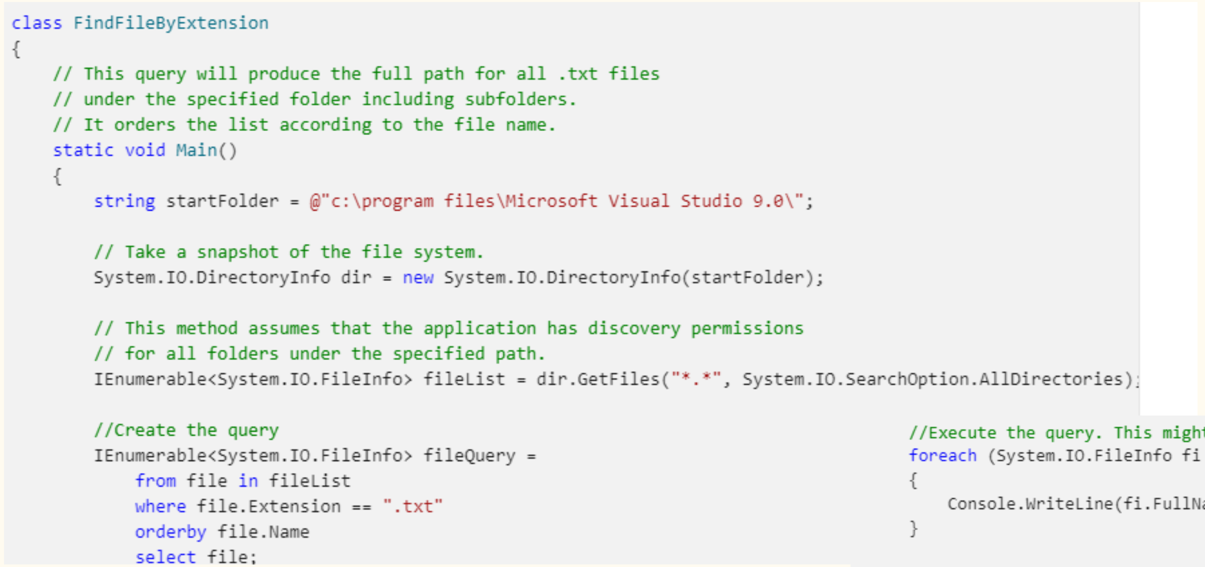
```
static void Cast()
{
    Plant[] plants = new Plant[] {
        new CarnivorousPlant { Name = "Venus Fly Trap", TrapType = "Snap Trap" },
        new CarnivorousPlant { Name = "Pitcher Plant", TrapType = "Pitfall Trap" },
        new CarnivorousPlant { Name = "Sundew", TrapType = "Flypaper Trap" },
        new CarnivorousPlant { Name = "Waterwheel Plant", TrapType = "Snap Trap" }
    };

    var query = from CarnivorousPlant cPlant in plants
                where cPlant.TrapType == "Snap Trap"
                select cPlant;

    foreach (Plant plant in query)
        Console.WriteLine(plant.Name);

    /* This code produces the following output:

        Venus Fly Trap
        Waterwheel Plant
    */
}
```

Scenario: Find the set difference between two lists

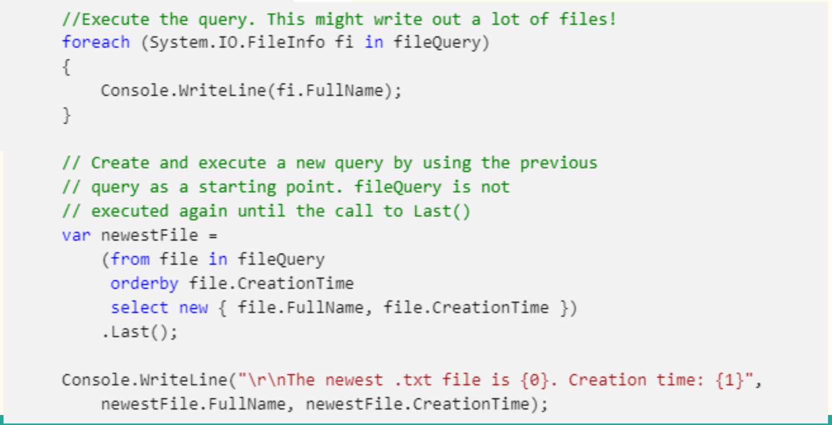


```
class CompareLists
{
    static void Main()
    {
        // Create the IEnumerable data sources.
        string[] names1 = System.IO.File.ReadAllLines(@"../../../names1.txt");
        string[] names2 = System.IO.File.ReadAllLines(@"../../../names2.txt");

        // Create the query. Note that method syntax must be used here.
        IEnumerable<string> differenceQuery =
          names1.Except(names2);

        // Execute the query.
        Console.WriteLine("The following lines are in names1.txt but not names2.txt");
        foreach (string s in differenceQuery)
            Console.WriteLine(s);

        // Keep the console window open until the user presses a key.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
     The following lines are in names1.txt but not names2.txt
    Potra, Cristina
    Noriega, Fabricio
    Aw, Kam Foo
    Toyoshima, Tim
    Guy, Wey Yuan
    Garcia, Debra
     */
```

Scenario: Query for files with a specified attribute



```
class FindFileByExtension
{
    // This query will produce the full path for all .txt files
    // under the specified folder including subfolders.
    // It orders the list according to the file name.
    static void Main()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*", System.IO.SearchOption.AllDirectories);

        //Create the query
        IEnumerable<System.IO.FileInfo> fileQuery =
            from file in fileList
            where file.Extension == ".txt"
            orderby file.Name
            select file;
```



```
//Execute the query. This might write out a lot of files!
foreach (System.IO.FileInfo fi in fileQuery)
{
    Console.WriteLine(fi.FullName);
}

// Create and execute a new query by using the previous
// query as a starting point. fileQuery is not
// executed again until the call to Last()
var newestFile =
    (from file in fileQuery
     orderby file.CreationTime
     select new { file.FullName, file.CreationTime })
    .Last();

Console.WriteLine("\r\nThe newest .txt file is {0}. Creation time: {1}",
    newestFile.FullName, newestFile.CreationTime);
```

Thread

- Thread
  - Thread & Process
  - Thread Class
  - Thread Life Cycle
  - Thread Problem / Thread Safety
  - Thread Synchronization
  - Dead Lock

Thread & Process

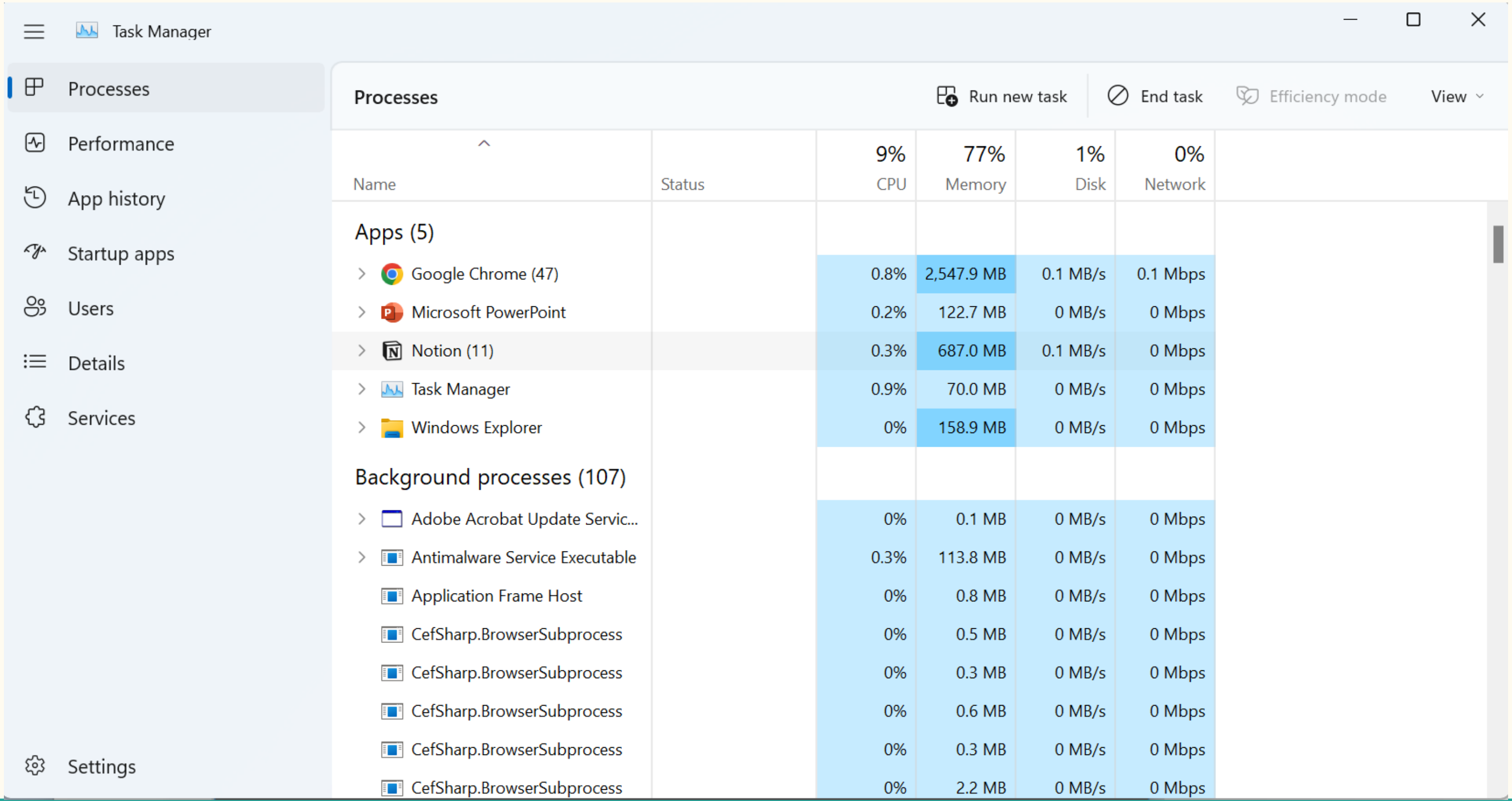
Process

- A process has a self-contained execution environment.

A process generally has a complete, private set of basic runtime resources; in particular, each process has its own memory space.

- Processes are often seen as synonymous with programs or applications However, what the user sees as a single application may in fact be a set of cooperating processes.
  - Eg.To facilitate communication between processes, most operating systems support *Inter Process Communication* (IPC) resources, such as pipes and sockets

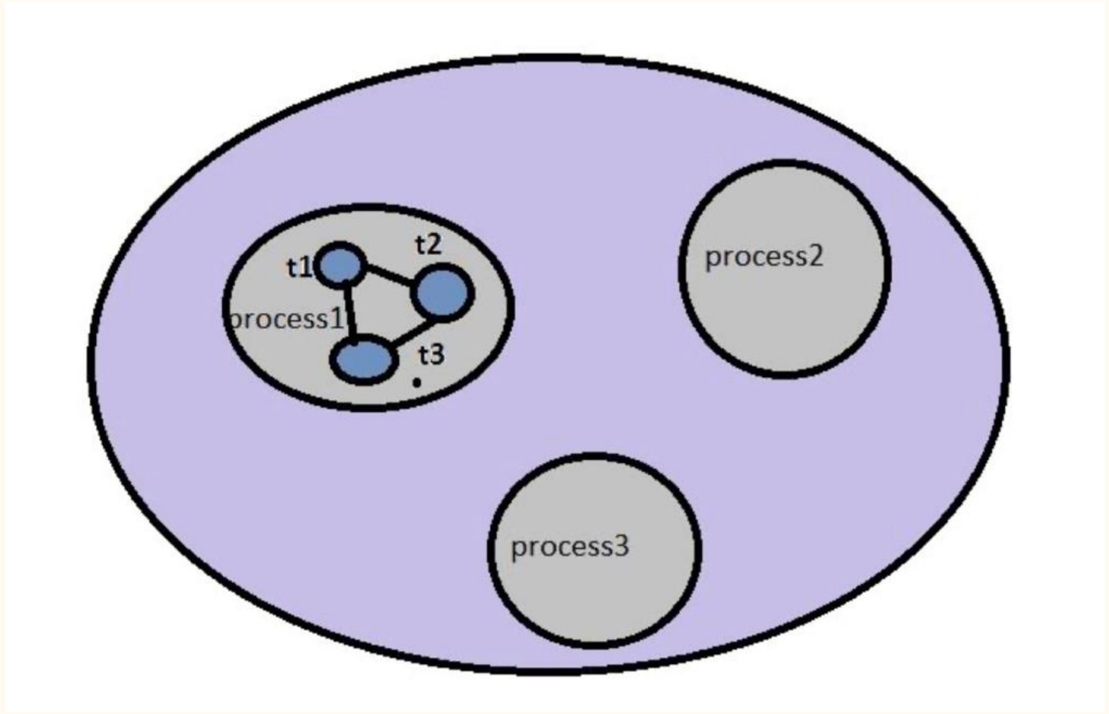
Process



Processes

Name	Status	9% CPU	77% Memory	1% Disk	0% Network
Apps (5)					
Google Chrome (47)		0.8%	2,547.9 MB	0.1 MB/s	0.1 Mbps
Microsoft PowerPoint		0.2%	122.7 MB	0 MB/s	0 Mbps
Notion (11)		0.3%	687.0 MB	0.1 MB/s	0 Mbps
Task Manager		0.9%	70.0 MB	0 MB/s	0 Mbps
Windows Explorer		0%	158.9 MB	0 MB/s	0 Mbps
Background processes (107)					
Adobe Acrobat Update Servic...		0%	0.1 MB	0 MB/s	0 Mbps
Antimalware Service Executable		0.3%	113.8 MB	0 MB/s	0 Mbps
Application Frame Host		0%	0.8 MB	0 MB/s	0 Mbps
CefSharp.BrowserSubprocess		0%	0.5 MB	0 MB/s	0 Mbps
CefSharp.BrowserSubprocess		0%	0.3 MB	0 MB/s	0 Mbps
CefSharp.BrowserSubprocess		0%	0.6 MB	0 MB/s	0 Mbps
CefSharp.BrowserSubprocess		0%	0.3 MB	0 MB/s	0 Mbps
CefSharp.BrowserSubprocess		0%	2.2 MB	0 MB/s	0 Mbps

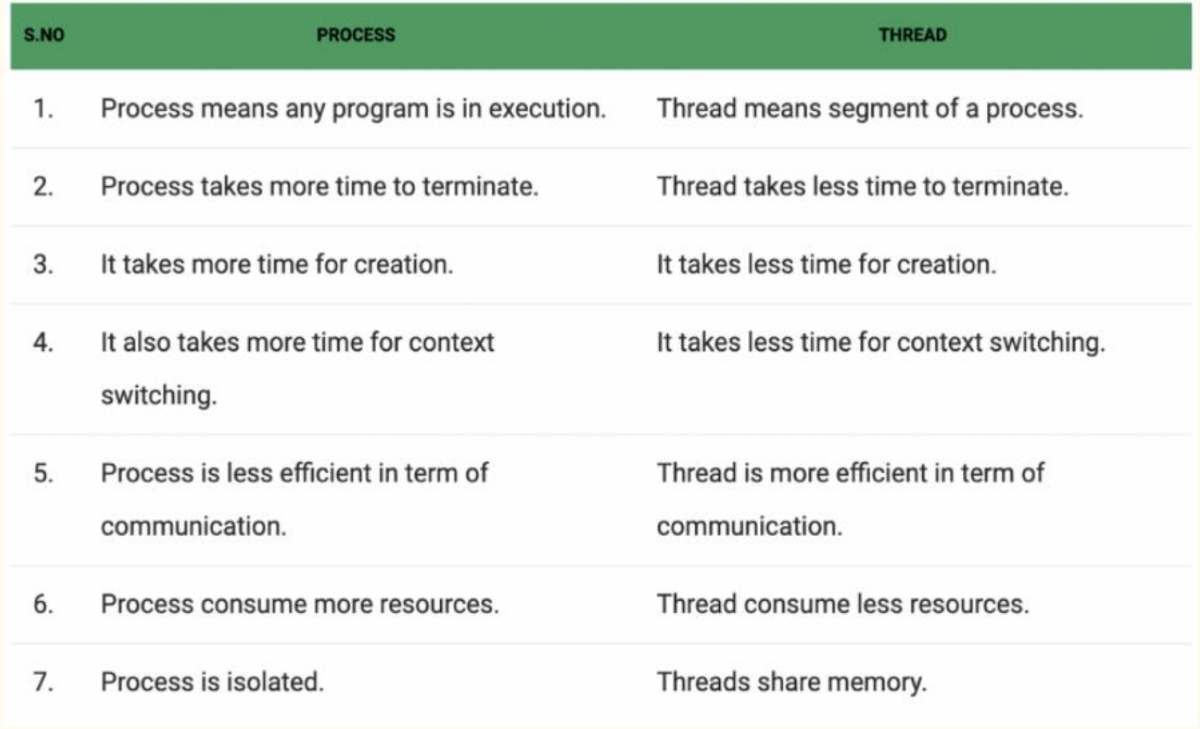
Thread vs. Process



Thread

- A thread is a lightweight sub-process, the smallest unit of processing. It has a separate path of execution.
- Threads are independent, if an exception occurs in one thread, it doesn't affect other threads.
- In other words, exceptions thrown in one thread cannot be handled by another thread.

Thread vs. Process



S.NO	PROCESS	THREAD
1.	Process means any program is in execution.	Thread means segment of a process.
2.	Process takes more time to terminate.	Thread takes less time to terminate.
3.	It takes more time for creation.	It takes less time for creation.
4.	It also takes more time for context switching.	It takes less time for context switching.
5.	Process is less efficient in term of communication.	Thread is more efficient in term of communication.
6.	Process consume more resources.	Thread consume less resources.
7.	Process is isolated.	Threads share memory.

Why Multi-Threading?

- To enhance parallel processing

- To reduce response time to the user
  - Many servers use multithreads to achieve high performance

- To utilize the idle time of the CPU
  - Unit testing uses threads to run test cases in parallel

- Prioritize your work depending on priority
  - Computer games is a good example of multi-threading process(loading maps when you are working on other things)

Thread Class

Thread Class

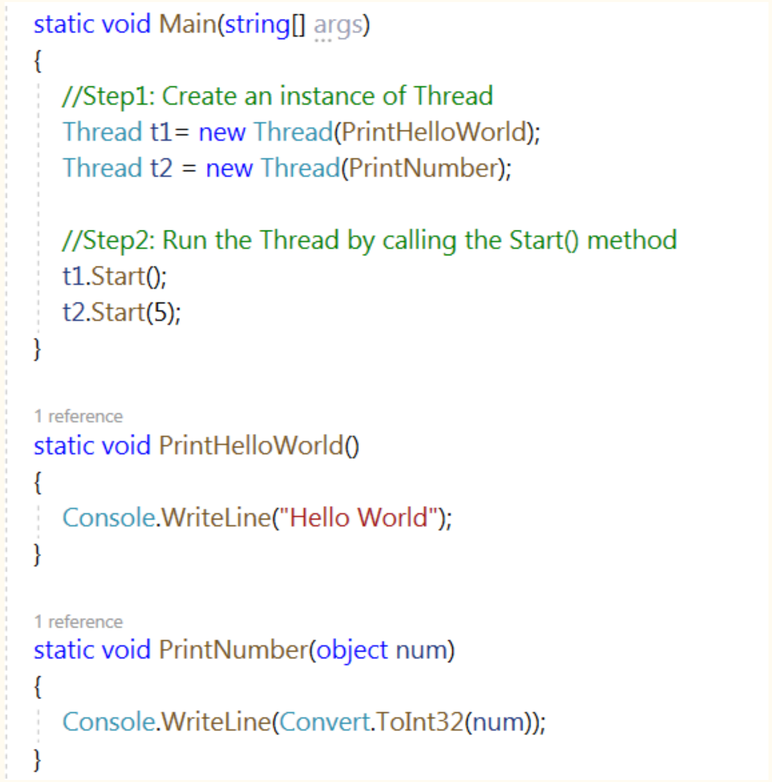
- In C#, a multi-threading system is built upon the Thread class, which encapsulates the execution of threads.
- This class contains several methods and properties which helps in managing and creating threads and this class is defined under System.Threading namespace.
- The first thread to be executed in a process is called the **main** thread.
- When a C# program starts execution, the main thread is automatically created. The threads created using the **Thread** class are called the child threads of the main thread.
- Programmers can always take control of the main thread.

Thread Class

- The thread class provides lots of properties. Some of the important properties are as follows:
- ***CurrentThread***: used to get the current running thread.
- ***Name***: used to get or set the name of the thread.
- ***Priority***: used to get or set the priority value(***Enum***) of the thread.
- ***ThreadState***: used to get the thread state value(***Enum***) of the thread.
- ***IsAlive***: returns a ***bool*** value representing whether or not this thread is alive
- ***IsBackground***: used to get or set the value(***bool***) indicating whether the thread is a background thread or not.

Thread Class

- Create a Thread



```
static void Main(string[] args)
{
    //Step1: Create an instance of Thread
    Thread t1= new Thread(PrintHelloWorld);
    Thread t2 = new Thread(PrintNumber);

    //Step2: Run the Thread by calling the Start() method
    t1.Start();
    t2.Start(5);
}

1 reference
static void PrintHelloWorld()
{
    Console.WriteLine("Hello World");
}

1 reference
static void PrintNumber(object num)
{
    Console.WriteLine(Convert.ToInt32(num));
}
```

Thread Life Cycle

The life cycle of a thread starts when an object of the Thread class is created and ends when the thread is terminated or completes execution.

- **The Unstarted State** − It is the situation when the instance of the thread is created but the Start method is not called.

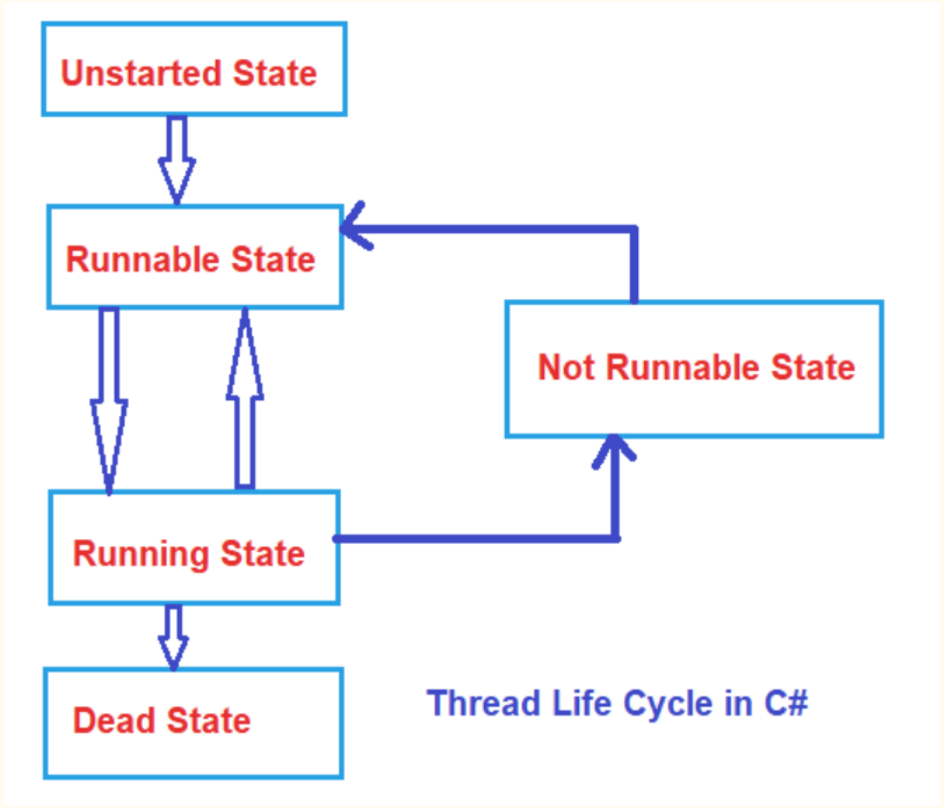
- **The Runnable State** − It is the situation when the thread is ready to run(the Start method has been called).

- **The Running State** – The thread is running and not stopped yet.

- **The Not Runnable State** − A thread is not executable, when
  - Sleep method has been called
  - Join method has been called
  - Blocked by I/O operations

- **The Dead State** − It is the situation when the thread completes execution or is aborted by calling Abort() method.

Thread Life Cycle



Not Runnable State

- Sleep() – Thread.Sleep() causes the current thread to suspend execution for a specific milliseconds. Sleep() is a static method in Thread class.

- Join() – The Join() method blocks the calling thread until the thread represented by this instance terminates.
    - Example: We create the Thread t in the main thread;
    - Calling t.Join() will cause the main thread to wait for thread t.

- Suspend() – Suspend() method is called to suspend the thread.（deprecated）

- Resume() – Resume() method is called to resume the suspended thread.(deprecated)

Types of Thread

Types of Thread

- In C# we can create two types of threads in the application, they are:
    - Foreground Thread
    - Background Thread

Foreground Thread vs. Background Thread

- Foreground threads are those threads that keep running even after the main application exits or quits. So, the foreground threads do not care whether the main thread is alive or not, it completes only when it finishes its assigned work. That means the life of a foreground thread does not depend upon the main thread. Foreground thread is the ***default type*** when a new thread is created.

- Background Threads are those threads that will quit if our main thread is finished. The life of a background thread depends on the main thread.
  - A thread can be changed to a background thread at any time by setting it's ***IsBackground*** property to ***true***.

Thread Safety

Thread Safety

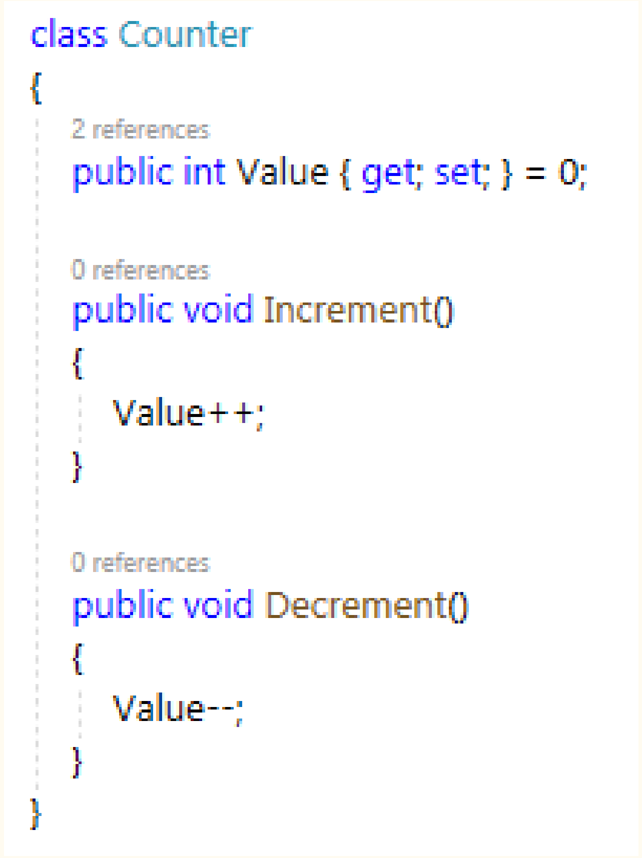
- Threads communicate primarily by sharing access to the same recourses such as fields or references to objects.

- This form of communication is extremely efficient, but it also makes some problems possible: *thread interference* and *data inconsistency*.

Thread Interference

- Consider a situation where two thread is operating on the same object at the same time.

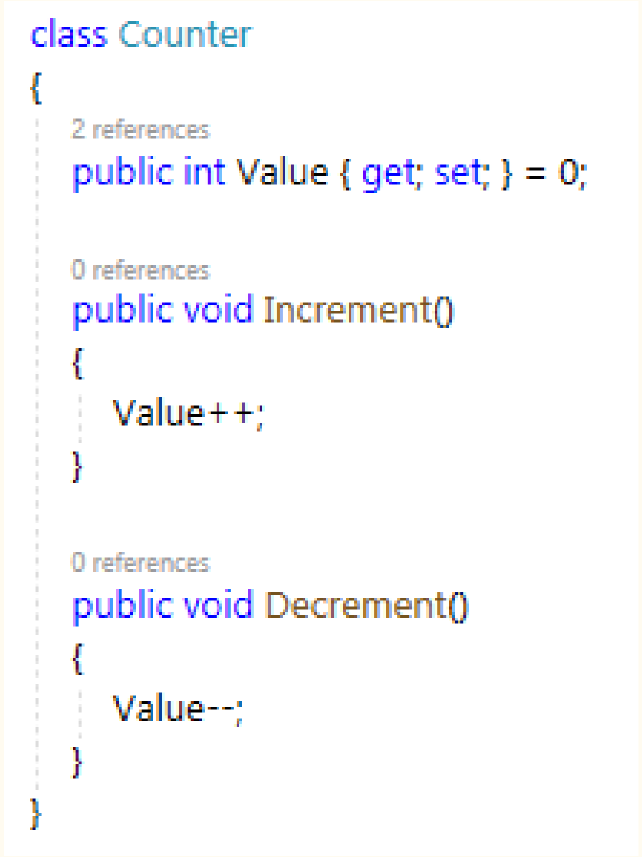
- Interference happens when two operations, running in different threads, but acting on the same data, *interleave*. This means that the two operations consist of multiple steps, and the sequences of steps overlap.



```
class Counter
{
    2 references
    public int Value { get; set; } = 0;

    0 references
    public void Increment()
    {
        Value++;
    }

    0 references
    public void Decrement()
    {
        Value--;
    }
}
```

Thread Interference

The App will decompose the Increment method into following steps:
• Retrieve the current Value.
• Increment the retrieved value by 1.
• Store the incremented value back in the Property.
What if thread A is calling increment and thread B is calling decrement? (What will be the result?)
This situation is also called the Race Condition



```
class Counter
{
    2 references
    public int Value { get; set; } = 0;

    0 references
    public void Increment()
    {
        Value++;
    }

    0 references
    public void Decrement()
    {
        Value--;
    }
}
```

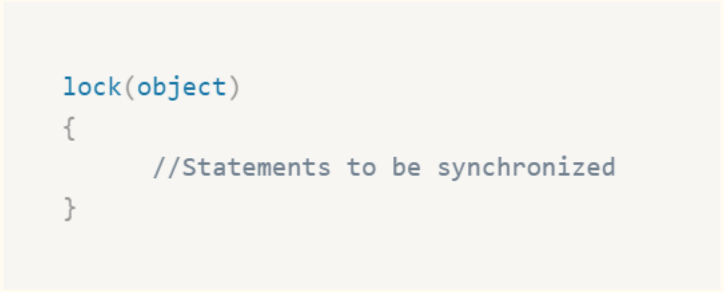
Data Inconsistency

- Data Inconsistency when different threads have inconsistent views of what should be the same data

- Eg. SameCounter class as before—Thread A increases the counter by 1, and thread B tries to print the value of counter at the same time. Now the value can either be 1 or 0

Thread Synchronization

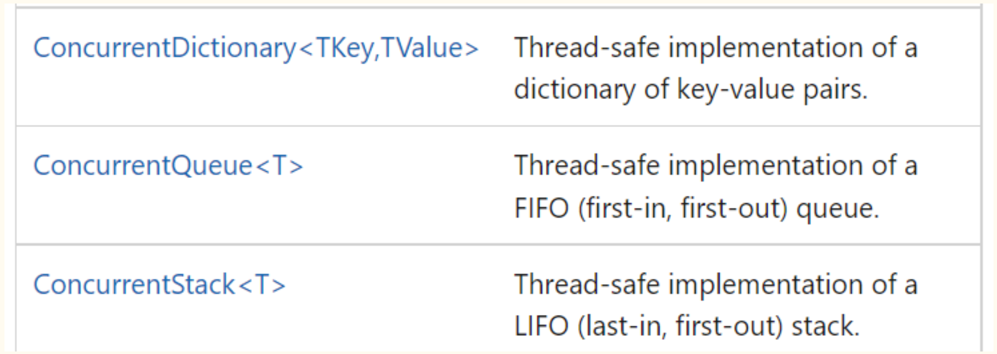
- Synchronization can be achieved by using the ***lock*** keyword.
- It is used lock the object and only allow one thread to access the locked object, execute the task and then the lock will be released.
- It ensures that other thread does not interrupt the execution until the execution finish.
- The lock can only apply on objects.
- Syntax:



```
lock(object)
{
      //Statements to be synchronized
}
```

Thread-Safe Collections

- The following table list some thread-safe collections under System.Collections.Concurrent namespace:

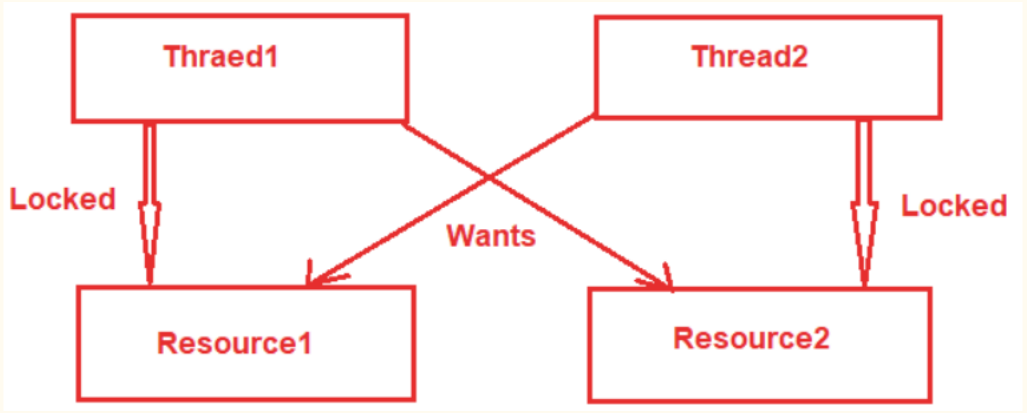


ConcurrentDictionary<TKey,TValue>	Thread-safe implementation of a dictionary of key-value pairs.
ConcurrentQueue<T>	Thread-safe implementation of a FIFO (first-in, first-out) queue.
ConcurrentStack<T>	Thread-safe implementation of a LIFO (last-in, first-out) stack.

Deadlock

Deadlock

- *Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other to release the lock.



Recap

- LINQ
  - What is LINQ?
  - LINQ to Objects
  - Standard Query Operators
  - LINQ to SQL

- Thread
  - Thread & Process
  - Thread Class
  - Thread Life Cycle
  - Thread Problem / Thread Safety
  - Thread Synchronization
  - Dead Lock

Any Questions?